Using XLang IPC to Enforce GPL 3.0 Process Boundaries in High-Performance Streaming Systems

Author: Shawn Xiong **Keywords:** IPC, GPL 3.0, process isolation, shared memory, LRPC, real-time streaming, remote objects, C++ embedding

---

Abstract

Integrating GPL 3.0–licensed components into proprietary systems demands a careful architecture that avoids creating a single combined work. The accepted strategy is to run GPL and proprietary modules as **separate processes** connected via **inter-process communication (IPC)**. This paper presents **XLang IPC**—a transparent, shared-memory–first IPC that treats **remote objects as if they were local**—and shows how it can serve as a clean, high-performance **process boundary** between GPL components and closed-source modules. We compare XLang IPC to OS-native IPC and gRPC, outline its control/data-plane design, and provide code for C++ API exposure and XLang script usage. A case study (GWS host-side streaming with Sunshine) demonstrates how to preserve GPL compliance without sacrificing low latency and throughput.
*Note: This paper describes engineering practices and is **not legal advice**.*

---

1. Introduction

GPL 3.0 imposes strong copyleft obligations on **derivative works**. When proprietary code dynamically or statically links to GPL code in the **same process**, the proprietary code may be treated as a derivative work and required to be distributed under GPL. A well-established compliance technique is to keep GPL and proprietary parts as **independent programs** and communicate **only via IPC**.

Modern streaming products need to bridge **high-rate binary payloads** (frames, tensors) and **low-latency control** (session start/stop, configuration). Generic IPC stacks often force trade-offs between developer ergonomics and performance. **XLang IPC** eliminates much of that friction by letting developers **import remote modules** with one line and automatically switching to **zero-copy shared memory** for large transfers, while using a lightweight RPC control plane for methods, properties, and events.

---

2. Background and Goals

XLang is a dynamic language and runtime oriented toward AI/IoT workloads. The IPC layer follows the same philosophy: **simple code, high performance, and transparent distribution**. In this paper we pursue three goals:

1. **Compliance:** Maintain a **hard process boundary** between GPL and proprietary modules.
2. **Performance:** Achieve near–memory-bandwidth throughput for bulk data; sub-millisecond control operations.
3. **Simplicity:** Enable **one-line imports** for remote objects and minimal ceremony for C++ API exposure.

---

3. Alternatives and Trade-offs

3.1 OS-Native IPC (pipes, domain sockets, named pipes, shared memory)

Pros: Maximum control; can be very fast. **Cons:** High engineering cost—message framing, schema evolution, error handling, backpressure, and eventing must be hand-built. Each new API requires editing both sides' transport logic.

3.2 gRPC (HTTP/2 + Protobuf)

Pros: Mature, cross-platform, strong tooling. **Cons:** Requires `.proto` schemas, codegen, and build integration; serialization adds CPU overhead and prevents **zero-copy** for multi-MB/GB payloads. Mapping rich **object semantics** (events/properties) requires extra scaffolding.

3.3 High-Level Convenience IPC (e.g., Python MP, REST)

Pros: Quick to bring up; batteries included. **Cons:** Serialization and copying dominate for large binary data; not suitable for real-time streaming in a C++-heavy codebase.

---

4. XLang IPC Overview

XLang IPC splits communication into **two planes**:

- **Control Plane (LRPC):** Compact binary envelopes for method calls, property gets/sets, and event notifications.
- **Data Plane (Shared Memory):** Memory-mapped regions for bulk payloads with a fixed-size **slice protocol** (e.g., 1 MiB per slice) and **event-driven backpressure**.

Programming model:

```
import stream_srv thru 'lrpc:9090' as stream
stream.Controller().start_session()
```

Remote objects feel local—object introspection, method invocation, and event subscription work without writing transport code. The runtime marshals small arguments over LRPC and places large buffers in shared memory, yielding **zero-copy** read/write paths where safe.

---

5. C++ Embedding and API Exposure

Before the code, here is a concise **A/B mental model** you can keep in mind when integrating XLang IPC.

Process A (GPL-side) *What it typically is:* the GPL component (e.g., an **image filter executable**). *Role:* **apply filters/transform images; emit stats/telemetry**. *IPC posture:* runs as an **independent executable**; embeds XLang and **publishes a narrow, documented surface** (methods, properties, events). Opens an LRPC endpoint (e.g., `lrpc:9090`, bound to localhost by default). Exposes classes such as `Images::Image`, `ImageFilter`, `FilterStats`, and data exporters.

Process B (Proprietary-side) *What it typically is:* the **user module** / orchestrator/UI/policy engine (e.g., UI app, session control, auth). *Role:* own user/session lifecycle; start/stop A; set policies; present UX. *IPC posture:*

embeds XLang, **imports A's surface**, and may also **publish its own callbacks/events** (e.g., `Window`, `PolicyManager`) so A can signal status. *IPC posture:* embeds XLang, **imports A's surface**, and may also **publish its own callbacks/events** (e.g., `Window`, `PolicyManager`) so A can signal status.

Startup & Handshake (recommended)

1. B launches or discovers A.
2. A binds its LRPC endpoint and publishes a **versioned API** (e.g., `ApiVersion`, `Capabilities`).
3. B performs `import img_srv thru 'lrpc:9090' as imgs`, probes version/capabilities, and wires event handlers.
4. (Optional) B publishes a callback object; A imports it for notifications.
5. Large transfers rendezvous on the **shared-memory data plane** only when needed; control stays on LRPC.

Threading & Safety

- XLang IPC is **event-driven**; callbacks execute on the runtime thread—offload long work to worker threads.
- Avoid blocking in event handlers; prefer async patterns or short critical sections.
- Pass **POD values or buffer handles** across the boundary; keep raw pointers private to a process.

Security Notes

- Bind LRPC to **localhost** unless cross-host is required; apply allowlists/ACLs.
- Restrict shared-memory regions to the user/session running A and B.
- Validate `ApiVersion` at import time and gate features accordingly.

Both processes embed the XLang runtime and **publish C++ classes**. This allows **methods, properties, and events** to cross the process boundary with no custom IPC code.

5.1 Image Processing Example (C++)

```
namespace Images {
  class Image {
    std::string m_url;
    Image* m_pimpl = nullptr;
  public:
    BEGIN_PACKAGE(Image)
      APISET().AddFunc<1>("to_tensor", &Image::To_Tensor, "image.to_tensor()");
      APISET().AddVarFunc("from_tensor", &Image::From_Tensor,
"image.from_tensor()");
      APISET().AddFunc<0>("save", &Image::Save, "image.save()");
    END_PACKAGE

    Image() {}
    Image(std::string url);
    bool Init();
    ~Image() { if (m_pimpl) delete m_pimpl; }

    virtual bool Save() { return m_pimpl ? m_pimpl->Save() : false; }
    virtual X::Value To_Tensor(int pixelFmt) {
```

```
      return m_pimpl ? m_pimpl->To_Tensor(pixelFmt) : X::Value();
    }
    virtual bool From_Tensor(X::XRuntime* rt, X::XObj* pContext,
                             ARGS& params, KWARGS& kwParams, X::Value& retValue) {
      return m_pimpl ? m_pimpl->From_Tensor(rt, pContext, params, kwParams,
retValue) : false;
    }
  };
}
```

5.2 UI Control Example (C++)

```
class Window : public ControlBase {
public:
  BEGIN_PACKAGE(Window)
    ADD_BASE(ControlBase);
    APISET().AddEvent("OnDraw");
    APISET().AddEvent("OnSize");
    APISET().AddClass<0, Button, Window>("Button");
    APISET().AddFunc<4>("CreateChildWindow", &Window::CreateChildWindow);
    APISET().AddFunc<1>("SetMenu", &Window::SetMenu);
  END_PACKAGE

  Window(Window* parent) : ControlBase(parent) {}
  Window() : ControlBase(nullptr) {}
};
```

5.3 Cross-Process Usage (XLang Script)

```
# Process A imports APIs exposed by Process B
import ui_srv thru 'lrpc:9091' as ui
win = ui.Window()
win.OnDraw += handle_draw

# Process B imports APIs exposed by Process A
import img_srv thru 'lrpc:9090' as imgs
T = imgs.Image("input.jpg").to_tensor()
```

Developers **do not** write socket code, message framing, or serializers—XLang IPC handles control messages and promotes large payloads to shared memory automatically.

---

6. Data-Plane Design: Slices and Backpressure

For large payloads (images, tensors, video frames), XLang uses memory-mapped regions and a **slice protocol**:

- Sender publishes `{total_length, slice_size}` on the control plane and signals **ready**.

- Receiver fetches slices in order, processes each, and **acks** to apply backpressure.
- Lease semantics ensure writers do not tear buffers while readers hold references.

This approach delivers **predictable throughput** and **bounded latency**, avoiding head-of-line blocking and excessive buffering.

---

7. Benchmarking — Methodology & Modeled Results

This section provides (a) a clear test plan and (b) **modeled estimates** for a demanding scenario you requested: **4K @ 240 FPS** frame transfer, both **raw** and **H.265-encoded**. These figures are **engineering estimates**, not measured benchmarks. Assumptions are listed so readers can reproduce or adjust.

7.1 Test Plan (for future measurement)

- **Workloads:** (W1) 1 GiB bytes in 1 MiB slices; (W2) float32/int8 tensors; (W3) control-heavy mixed calls.
- **Metrics:** throughput (MB/s), end-to-end latency, CPU%, and copy count.
- **Procedure:** warm-up; ≥5 trials; pin server/client to separate cores; release builds; minimize background load.

7.2 Assumptions for Modeled Estimates

- **Host:** 2024 desktop/workstation CPU with AVX2/AVX-512; sustained single-thread memcpy ≈ **20 GB/s**.
- **Copy model (XLang IPC):** producer copies into shared memory; consumer materializes into stub memory → **2 copies per frame** (proxy → SHM; SHM → stub).
- **Copy model (OS pipes/domain sockets):** user→kernel→user per hop → **≈4 copies per frame**.
- **Copy model (gRPC over loopback):** HTTP/2 + protobuf framing → **≥4 copies + serialization** (effectively worse than pipes for large payloads).
- **Python MP Queue:** pickling + copies → **high overhead**.
- **Arrow streaming:** batch framing + (de)serialization; memory-mapped reads can be fast but add encode/decode work.
- **Frame geometry:** 4K = 3840×2160 = 8,294,400 pixels.
  - **NV12 (4:2:0, 8-bit):** 1.5 B/px → **12.4416 MB/frame**.
  - **RGB8 (4:4:4, 8-bit):** 3 B/px → **24.8832 MB/frame**.
  - **RGB10 (approx packed):** ~4 B/px → **33.1776 MB/frame**.
- **Frame period @ 240 FPS: 4.167 ms**.

> **Copy time per frame (two-copy path):** `t_ms ≈ (frame_MB × copies / 20,000) × 1000`.

7.3 Modeled: 4K240 RAW (NV12, 12.44 MB/frame)

- **Base data rate (no copies):** 12.4416 MB × 240 FPS ≈ **2,986 MB/s ≈ 2.85 GiB/s**.
- **XLang IPC (2 copies):** copy time per frame ≈ **1.244 ms** (0.622 ms × 2).

  - **Budget left in 4.167 ms:** ~**2.92 ms** for capture/encode/scheduling.
  - **Estimated sustainable FPS ceiling (mem-bound):** `20,000 MB/s ÷ (12.4416 MB × 2) ≈ **804 FPS**` (practical lower due to other work).
  - **CPU load (per side):** copies use **~30–40% of one core** at this rate; control plane <0.1 ms/frame.

- **OS pipes/domain sockets (~4 copies):** copy time ≈ **2.488 ms** → budget left ~**1.68 ms** → likely **borderline** under load.

- **gRPC (HTTP/2, protobuf):** serialization + kernel copies; expected max **≲1–2 GiB/s** on loopback with very high CPU → **insufficient** for 2.85 GiB/s raw stream at 240 FPS.

- **Python MP Queue:** pickling dominates; **not feasible** at this rate.

- **Arrow stream:** 1–3 GiB/s typical on fast SSD/mmaps with batch encode/decode; **borderline/insufficient** for 2.85 GiB/s + latency budget.

Table A — 4K240 RAW (NV12) — Modeled

IPC method	Copies/frame	Copy time (ms)	Data rate needed	Meets 240 FPS?	Notes
XLang (SHM)	2	**1.244**	2.85 GiB/s	**Yes**	Leaves ~2.9 ms for other work; control overhead ~tens of µs
OS pipes / domain sockets	~4	**2.488**	2.85 GiB/s	**Maybe**	Tight headroom; kernel copies + context switches
gRPC (loopback)	≥4 + serde	>3 ms eqv.	2.85 GiB/s	**No**	HTTP/2 framing + protobuf; CPU heavy
Python MP Queue	many + pickle	>10 ms	2.85 GiB/s	**No**	Not intended for this workload
Arrow streaming	3–4 + batch	3–6 ms	2.85 GiB/s	**No**	Batch encode/decode adds latency

7.4 Modeled: 4K240 RAW (RGB8, 24.88 MB/frame)

- **Base data rate:** 24.8832 MB × 240 ≈ **5,973 MB/s ≈ 5.57 GiB/s**.
- **XLang (2 copies):** copy time ≈ **2.488 ms** → budget left ~**1.68 ms** (feasible but tighter).
- **OS pipes (~4 copies):** copy time ≈ **4.976 ms** → **misses** 4.167 ms frame budget.

Table B — 4K240 RAW (RGB8) — Modeled

IPC method	Copies/frame	Copy time (ms)	Data rate needed	Meets 240 FPS?	Notes
XLang (SHM)	2	**2.488**	5.57 GiB/s	**Maybe**	Requires optimized memcpy (AVX-512), pre-allocated buffers

IPC method	Copies/frame	Copy time (ms)	Data rate needed	Meets 240 FPS?	Notes
OS pipes / domain sockets	~4	**4.976**	5.57 GiB/s	**No**	Exceeds frame budget before other work
gRPC / MP / Arrow	many	>5 ms	5.57 GiB/s	**No**	Serialization costs dominate

7.5 Modeled: 4K240 H.265 Encoded

Assume **H.265** with low-latency preset at **~160 Mbit/s** (typical for 4K high-FPS game/desktop content; actual values vary **60–300 Mbit/s** by quality/GOP).

- **Per-second IPC payload: 20 MB/s**.
- **Per-frame payload:** 20 MB/s ÷ 240 ≈ **0.083 MB/frame**.
- **XLang (2 copies):** copy time ≈ **0.008 ms/frame** → **negligible**.
- **All other IPCs:** also negligible at this bitrate; the bottleneck is **encoder latency**, not IPC.

Table C — 4K240 H.265 Encoded — Modeled

IPC method	Copies/frame	Copy time (ms)	Data rate needed	Meets 240 FPS?	Bottleneck
XLang (SHM)	2	**0.008**	20 MB/s	**Yes**	Encoder settings (GOP/latency) dominate
OS pipes / gRPC / MP / Arrow	≥2	0.01–0.1	20 MB/s	**Yes**	IPC not the limiting factor here

8. Case Study: Host-Side Streaming with a GPL Component (GWS + Sunshine)

In my role as **Principal Software Architect at Tencent North America**, I led the architecture of the **Remote Graphics Workstation (GWS)** (cloudstudio.games/products/gws). On the host side, the system integrates the **GPL 3.0-licensed Sunshine** streaming server for high-performance capture/encoding. To preserve GPL compliance, Sunshine runs as a **separate process**. Proprietary host modules (session management, authentication, orchestration) run in another process. The two communicate only via **XLang IPC**.

Why this satisfies the boundary requirement

- **No linking:** GPL and proprietary code do not share a process or link against each other.
- **Clear IPC interface:** Documented RPC + shared-memory channels; each side is a usable program on its own.
- **High performance:** Zero-copy slices sustain real-time streaming needs.

Example host-side control

```
import sunshine_srv thru 'lrpc:9090' as sunshine
sunshine.StreamController().StartSession(display_id=0)
```

Example telemetry

```
stats = sunshine.EncoderStats()
print(stats.bitrate, stats.latency_ms)
```

This achieves three objectives simultaneously: **legal isolation**, **minimal integration effort**, and **low-latency operation**.

---

9. Developer Workflow Checklist

1. **Split processes:** Package GPL and proprietary modules as distinct executables.
2. **Embed runtime:** In each process, embed XLang and **register** C++ APIs (methods, properties, events).
3. **Start listeners:** GPL side listens on an **LRPC** port; proprietary side imports and calls.
4. **Bulk data:** For large payloads, rely on XLang's **shared-memory slices**.
5. **Events & backpressure:** Use events for readiness/ack; avoid polling.
6. **Security:** Apply OS ACLs to shared memory; authenticate/control which ports may connect.
7. **Testing:** Reproduce benchmarks with W1–W3; record throughput/latency/CPU; validate no accidental in-process linking paths exist.

---

10. Security and Isolation Considerations

- **Shared-memory ACLs:** Restrict region creation/mapping to authorized users.
- **Port hygiene:** Bind the LRPC listener to appropriate interfaces; consider local-only bindings for host-internal flows.
- **Timeouts & failure modes:** Ensure bounded waits; handle disconnects and stale leases.
- **Schema evolution:** Keep control messages backward-compatible; expose explicit version fields if needed.

---

11. Limitations and Future Work

- **Cross-host fast-path:** Shared-memory zero-copy is local-only; use network transports for remote hosts; explore RDMA/GPUDirect for future zero-copy over NIC.
- **Richer schema interop:** Consider mapping Arrow arrays directly to shared regions in mixed stacks to eliminate re-serialization.
- **Hardening:** Optional encryption/auth for the control plane; per-session keys for shared-memory rendezvous.

---

12. Conclusion

XLang IPC provides a uniquely simple and fast way to enforce **GPL 3.0 process boundaries** in streaming systems. It delivers **transparent remote objects**, a **lightweight control plane**, and a **zero-copy shared-memory data plane**—so teams can isolate GPL components from proprietary modules **without** giving up performance or burning engineering time on bespoke IPC. The GWS + Sunshine case demonstrates that the approach scales to demanding real-time workloads while keeping compliance clean.

Appendix A — Reference Snippets

XLang server/client (bytes in slices): Use an XLang file that registers a remote object, prepares a large buffer, and serves slices to a client loop. **Python MP baseline:** Simple Queue-based server/client transferring 1 GiB in 1 MiB slices. **Arrow IPC baseline:** Record-batch stream written to a file; client memory-maps and iterates.

Appendix B — Result Tables (Modeled, based on §7)

B.1 1 GiB / 1 MiB Slices (throughput sanity, modeled)

Workload	Method	Bytes (GiB)	Slice (MiB)	Time (s)	Throughput (MB/s)	Notes
W1	XLang IPC	1.0	1	~0.12–0.20	**5,000–8,500**	SHM + low control overhead
W1	Python MP (Queue)	1.0	1	~1.5–3.5	**300–700**	Pickle + copies
W1	Arrow IPC (Stream)	1.0	1	~0.4–0.9	**1,100–2,500**	Batch encode/decode

B.2 4K240 Raw NV12 (12.44 MB/frame)

Method	Copies	Copy time (ms/frame)	Data rate	Meets 240 FPS?
XLang (SHM)	2	**1.244**	2.85 GiB/s	**Yes**
OS pipes	~4	**2.488**	2.85 GiB/s	**Maybe**
gRPC	≥4 + serde	>3	2.85 GiB/s	**No**

B.3 4K240 Raw RGB8 (24.88 MB/frame)

Method	Copies	Copy time (ms/frame)	Data rate	Meets 240 FPS?
XLang (SHM)	2	**2.488**	5.57 GiB/s	**Maybe**
OS pipes	~4	**4.976**	5.57 GiB/s	**No**
gRPC	≥4 + serde	>5	5.57 GiB/s	**No**

B.4 4K240 H.265 Encoded (~160 Mbit/s)

Method	Copies	Copy time (ms/frame)	Data rate	Meets 240 FPS?

Method	Copies	Copy time (ms/frame)	Data rate	Meets 240 FPS?
XLang (SHM)	2	**0.008**	20 MB/s	**Yes**
OS pipes / gRPC / MP / Arrow	≥2	0.01–0.1	20 MB/s	**Yes**